Received 6 February 2024, accepted 23 February 2024, date of publication 29 February 2024, date of current version 8 March 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3371499

RESEARCH ARTICLE

Compiler Provenance Recovery for Multi-CPU Architectures Using a Centrifuge Mechanism

YUHEI OTSUBO[1,2,3], **AKIRA OTSUKA**[3], **(Member, IEEE), AND MAMORU MIMURA**[3,4]
[1]National Police Academy, Fuchu, Tokyo 183-0003, Japan
[2]National Police Agency, Chiyoda, Tokyo 100-8974, Japan
[3]Institute of Information Security, Yokohama, Kanagawa 221-0835, Japan
[4]National Defense Academy, Yokosuka, Kanagawa 239-0811, Japan

Corresponding author: Yuhei Otsubo (dgs157101@iisec.ac.jp)

ABSTRACT Bit-stream recognition (BSR) has a wide range of applications, including forensic investigations, detecting copyright infringement, and analyzing malware. In order to analyze file fragments recovered by digital forensics, it is necessary to use a BSR method that can accurately classify classes while addressing various domains without preprocessing the raw input bitstream. For example, it is important to note that in the case of compiler provenance recovery, a type of BSR, the same bit sequence can have different meanings for different CPU architectures. As a result, traditional methods that rely heavily on disassembly tools, such as IDA Pro, may have limited in applicaballity scope to programs designed for specific CPU architecture. To address the aforementioned limitation, we proposed a novel learning method. Our method involves the upstream layers (sub-net) capturing global features and instructing the downstream layers (main-net) to shift focus, even when a portion of the input bit-stream has identical values. Through our experiments, we utilized a model that was less than 1/300 the size of the state-of-the-art model. Despite its smaller size, our method achieved the highest classification performance of 99.54 on a multi-CPU architecture, outperforming existing methods.

INDEX TERMS Binary analysis, compiler provenance recovery, machine learning, transfer learning, fine-tuning.

I. INTRODUCTION

Bit-stream recognition (BSR) is a classification task that takes a bit-stream as input and outputs a class label. In the field of cybersecurity, which is the practice of protecting systems, networks, and programs from digital attacks, various studies have used methods involving BSR models, including the classification of malicious programs [25], author identification of programs [36], discovering vulnerabilities in programs [28], function recognition [11], [19], [34], and recovering corrupted files [7]. Bit-streams sometimes appear to be very complicated, as they are mostly produced by artificial generative models such as program compilers and audio codecs. Because the bit-stream structure varies greatly depending on the artificial generative model and the location or context from which the bit-stream was taken, data engineering (DE) is among the most significant steps for achieving classification accuracy.

Compiler provenance recovery (CPR), one of the applications of BSR, refers to the task of identifying the environment in which given program binaries were created. When CPR is applied to malware analysis, it can provide important information for determining the author of the malware [27]. For the analysis of file fragments recovered by digital forensics, we need a BSR method that takes a raw input bitstream and enables highly accurate class classification while simultaneously addressing various domains without any preprocessing. However, there are still only a few studies that use raw bit-streams as training data and has high classification accuracy while simultaneously addressing various domains [6]. CPR is mainly a classification for machine language instruction sequences. However, because the model cannot be trained efficiently with raw bit-streams, various methods for creating feature vectors have

The associate editor coordinating the review of this manuscript and approving it for publication was Ines Domingues.

been proposed, including replacing disassembled machine language instructions with several categories of symbols to see the flow of program code [31] and creating feature vectors on a per-function basis [33], [35]. Otsubo et al. proposed o-glassesX [27], which can be trained with data that are relatively close to raw bit-streams, and then classification can be performed with high accuracy by dividing the bit-stream into instruction units in pre-processing.

Most existing BSR models are domain dependent and are designed to improve classification accuracy by narrowing down the input bit-stream target domain and specializing the DE for a particular purpose. Similarly in CPR, the key to highly accurate classification is to limit the corresponding CPU architecture and DE according to its type. There are few generic BSR methods exist that can apply existing models to other BSR problems without redesigning the DE to achieve high classification performance. Therefore, in this study, we aim to propose a CPR method without DE that can classify bit-streams with high accuracy.

It is also known that, in general, losses in natural language processing models decrease as the size of the model, the size of the data set, and the computational power used for training increase [10]. In CPR, there has been a trend toward the giant size of models, as seen in the publication of a method using RoBERTa [6]. On the other hand, there are issues such as increasing the size of the data set for training in accordance with the model size increase, and the increase in the computational cost of training limits the fields in which it can be applied. Therefore, our other goal is to propose a more lightweight and accurate classification method.

We propose a centrifuge mechanism in which the upstream sub-net transitions the input to a space corresponding to sub-labels. The key idea of the centrifuge mechanism is to utilize a sub-net predictor that captures global features to automate the process of narrowing down the target domain, which has been done manually in the past. The centrifuge mechanism consists of a sub-net that learns global features and a broadcast concatenator. The broadcast concatenator concatenates an input of the sub-net and the sub-net's output. This concatenation changes operations that use local features of the main net to those that consider global features.

The centrifuge allows the downstream main-net to focus on more difficult classifications. For the centrifuge mechanism, we checked the accuracy and characteristics of the main-net and sub-net predictions using various learning methods. One of them pre-trains the main-net using the sub-label's ground truth instead of the sub-net's output. This method was able to give the sub-net the role of sub-label prediction without using a loss function for sub-label classification. Additionally, we found that sub-predictions tend to be highly accurate when the sub-label classification contributes to the essence of the main prediction.

The contributions of this paper are as follows.

- Our proposed model is a lightweight CPR model able to simultaneously support multiple CPU architectures, and it has achieved cutting edge performance for CPR in terms of classification performance.
- We demonstrated that a single loss function can be used to generate a sub-net in the model that allow subclass classification independent of the main class.
- We proposed a new learning method that improves the interpretability of the output by explicitly assigning roles in the model.

II. RELATED WORK

There are few BSR models that use raw bit-streams as training data and has high classification accuracy has yet been developed. In this section, we describe existing work on CPR as an example of BSR problems. In this paper, we experimentally demonstrate our model's applicability to CPR.

A. COMPILER PROVENANCE RECOVERY FOR SINGLE-CPU ARCHITECTURE

Rosenblum et al. used a conditional random field (CRF [21]) and set up a classifier that took a bit-stream as input to identify one of the three compiler families with a 0.924 accuracy rate [35]. They disassembled the bit-stream with an IA-32 architecture and found a typical matching instruction pattern called "idioms" that predicted the compiler families. Their early result is only for the relatively easy compiler family identification with three classes. Rosenblum et al. have improved their method and proposed a tool named ORIGIN [33]. ORIGIN's linear support vector machine [4] takes the feature of an independent "function" as input and predicts both the optimization level and the version in addition to the compiler family, but it has difficulty identifying the compiler versions. ORIGIN's CRF takes the same features of multiple adjacent functions as input for the prediction and performs with a higher accuracy of 0.9 and above, despite the number of classes having increased from 3 to 18. However, the size of the input data required for predicting compiler provenance is larger.

Rahimian et al. developed BinComp [31], an approach in which the syntax, structure, and semantics of disassembled functions are analyzed to extract the compiler provenance. In their experiments, BinComp had an identification accuracy of 0.801 in 8-class classification.

Otsubo et al. proposed o-glasses [26], which separates machine instructions from other data with high precision. Yang et al. proposed BinEye [40], a method for identifying compiler optimization levels. These two methods achieve highly accurate classification by processing each instruction in a CNN. o-glasses targets the x86 architecture and uses a disassembler to decompose a byte sequence into a machine language instruction sequence. BinEye, on the other hand, is for the ARM architecture, and takes advantage of ARM's fixed-length instructions to achieve instruction-by-instruction processing by breaking them down into 4-byte units without disassembler.

o-glassesX [27] has succeeded in identifying compilers with very high accuracy by dividing the raw bit-stream into x86/x86-64 machine language instruction units and combining a convolutional neural network (CNN [15]) and an attention mechanism [18]. o-glassesX can identify the compiler family and the optimization level and version with high accuracy merely by analyzing instruction sequences without regard to functions.

Tian et al. proposed NeuralCI [38], which performs compiler estimation on a function-by-function basis, and NeuralCI performs instruction normalization, which replaces the address values of extracted machine instructions, and instruction embedding using skip-gram model [23]. There are two types of network models: CNN-based and GRU-based [2], [3], both of which can perform compiler estimation with high accuracy.

Benoit et al. proposed a graph neural network-based [41] compiler identification method named the site neural network (SNN) [1]. To attain scalability at the binary level, they made feature extraction simplified by forgetting almost everything in a binary code's structured control flow graph (CFG) [20] except transfer control instructions and performing a parametric graph reduction.

All of these existing studies require pre-processing, such as disassembly of machine language instructions and structural analysis of functions, to achieve their high precision classifications. We show that our method achieves high accuracy classification without pre-processing.

B. COMPILER PROVENACE RECOVERY FOR MULTI-CPU ARCHITECTURE

Several compiler identification tools are already available (i.e., IDA Pro[1] and PEiD.[2] These tools are roughly signature-based and typically rely on metadata or other details in the program header. Their exact matching algorithm may fail if even a slight difference between signatures is present, or if the header information has been stripped or is otherwise unavailable.

Pizzolotto and Inoue proposed the Optimization Detector [30], which uses CNN or LSTM [8], [9] to identify the compiler's optimization level. They checked the classification performance of binaries with two different compilers and five different optimization levels for each of seven different CPU architectures. As a result, they confirmed that the classification performance is better with raw byte strings than with encoded input data, provided that the input data is of a certain length. Their experimental results show the possibility of supporting multiple CPU architectures simultaneously by using raw input data, which is demonstrated by BinProv [6], described next.

He et al. propose BinProv, an end-to-end compilation provenance identification framework with the contextual semantics in binary code using RoBERTa [16]. BinProv is a compiler classification method that simultaneously supports multiple CPU architectures and shares the same objectives as our proposed method. In this paper, we compare the classification performance between BinProv and our method.

[1]https://www.hex-rays.com/products/ida/
[2]https://www.aldeid.com/wiki/PEiD

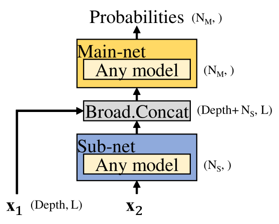


FIGURE 1. Outline of the centrifuge mechanism.

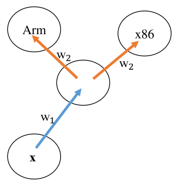


FIGURE 2. Transition of the input x by the first linear layer of the main-net.

III. CENTRIFUGE MECHANISM

The structure of bit-streams differs greatly depending on the type of file. Even if we consider only the case of machine language instruction strings, a general linear layer cannot transform the input bit-stream into the instruction vector sequence since bit-blocks with the same value have different meanings on different CPU architectures. We therefore propose a centrifuge mechanism that transforms local features like bit-blocks into a feature vector by considering global features.

Even bit-blocks with the same value can have different meanings if the CPU architecture is different. Assuming that the sub-net learns the global features such as the CPU architecture of the input bit-stream, we expect the output of the linear layer after sub-net embedding to produce a feature vector sequence similar to the instruction vector sequence.

A basic form of the centrifuge mechanism is shown in Figure 1. Any model can be set up as a main-net and sub-net. When the input of the sub-net is $\mathbf{x}_2$ and the sub-net model is Sub with parameter $\mathbf{w}_S$, the output $\mathbf{y}_S$ is expressed by

$$\mathbf{y}_S = \mathrm{Sub}(\mathbf{x}_2; \mathbf{w}_S). \tag{1}$$

When an operator $\oplus$ represents a broadcast concatenator operation, the input of the main-net is $\mathbf{x}_1 \oplus \mathbf{y}_S$. Then, the output $\mathbf{y}_M$ is expressed by

$$\mathbf{y}_M = \mathrm{Main}(\mathbf{x}_1 \oplus \mathbf{y}_S; \mathbf{w}_M), \tag{2}$$

where the main-net model is Main with parameter $\mathbf{w}_M$.

We assume that the first layer of the main-net is a linear layer. When the parameter of the linear layer is $\{\mathbf{w}_1, \mathbf{w}_2\}$, the output of the linear layer $\mathbf{x}'$ is given by

$$\mathbf{x}' = \mathrm{Linear}(\mathbf{x}_1 \oplus \mathbf{y}_S) = \mathbf{x}_1\mathbf{w}_1 + \mathbf{y}_S\mathbf{w}_2. \tag{3}$$

Then, assuming that the sub-net has acquired global features like the CPU architecture, $\mathbf{x}'$ is expected to be a transition of input x to the space for each global feature, a process shown in Figure 2 that resembles the action of a centrifuge.

$\mathbf{x}_2$ can take any value regardless of $\mathbf{x}_1$. In this paper, we refer to the centrifuge when $\mathbf{x}_2$ and $\mathbf{x}_1$ are equal as the self-centrifuge and when $\mathbf{x}_2$ and $\mathbf{x}_1$ are different as the source-target centrifuge.

For the sub-net to explicitly acquire global features such as the CPU architecture, either a loss function or a learning method needs to be devised. Next, we provide an overview of transfer learning, fine-tuning, and an objective function combining two loss functions.

A. TRANSFER LEARNING

Transfer learning [29] is a learning method that focuses on storing knowledge gained while solving a problem and applying it to a different but related problem.

1) UPSTREAM TRANSFER LEARNING: PRE-TRAINING SUB-NET

Empirically, models such as AlexNet that are trained on vast amounts of labeled data such as Imagenet learn generic features in the layer close to the input (upstream). These features are also effective in other tasks, so transferring the weights of this learned model can reduce training time and create a highly accurate model when there are little labeled data in the destination environment. A common approach to transfer learning is to work with a trained model as a feature extractor. We name this approach upstream transfer learning (UTL) to distinguish it from the downstream-focused approach described in the next section,

To apply UTL to our model, we pre-train the upstream sub-net to work them as feature extractors. When the ground truth for the subclass is $\mathbf{t}_\mathrm{S}$, the scheduled learning rate is α, the weight decay [12] coefficient is λ and the loss function for the subclass is $\mathcal{L}_\mathrm{S}$, the pre-learning of the sub-net is given by

$$\mathbf{w}_\mathrm{S} := \mathbf{w}_\mathrm{S} - \alpha\left(\frac{\partial \mathcal{L}_\mathrm{S}(\mathbf{y}_\mathrm{S}, \mathbf{t}_\mathrm{S})}{\partial \mathbf{w}_\mathrm{S}}\right) - \lambda \mathbf{w}_\mathrm{S}. \tag{4}$$

After completing the pre-training, the parameters of the main-net are updated using

$$\mathbf{w}_\mathrm{M} := \mathbf{w}_\mathrm{M} - \alpha\left(\frac{\partial \mathcal{L}_\mathrm{M}(\mathbf{y}_\mathrm{M}, \mathbf{t}_\mathrm{M})}{\partial \mathbf{w}_\mathrm{M}}\right) - \lambda \mathbf{w}_\mathrm{M}, \tag{5}$$

where $\mathbf{t}_\mathrm{M}$ is the ground truth for the main class and $\mathcal{L}_\mathrm{M}$ is the loss function for the main class.

2) DOWNSTREAM TRANSFER LEARNING: PRE-TRAINING MAIN-NET (PROPOSED METHOD)

In this section, we describe transfer learning focusing on downstream (downstream transfer learning (DTL)), in contrast to the upstream-focused approach in the previous section. When the upstream weights of the network are fixed, the upstream works as a feature extractor. In contrast, if we fix the weights downstream, near the exit layer of the network, then the downstream can be viewed as a loss function in complex formulas.

Algorithm 1 DTL Algorithm

```
Input: Data augmentation function DA, Loss function $\mathcal{L}_\mathrm{M}$,
    main-net predictor Main, sub-net predictor Sub, training
    dataset $S := \bigcup_{i=1}^{n}\{(\mathbf{x}_{1i}, \mathbf{x}_{2i}, \mathbf{t}_{\mathrm{S}i}, \mathbf{t}_{\mathrm{M}i})\}$, mini-batch size
    $b$, weight decay coefficient $\lambda$, scheduled learning rate $\alpha$,
    initial weight $\mathbf{w}_0$, epochs $e$.
Output: Trained weight $\mathbf{w} = \{\mathbf{w}_\mathrm{S}, \mathbf{w}_\mathrm{M}\}$
 1: Initialize weight $\mathbf{w} = \mathbf{w}_0$.
    {Main-net training}
 2: for $i = 1 \ldots e$ do
 3:   while not converged do
 4:     Sample a mini-batch $B$ of size $b$ from $S$.
 5:     $\mathbf{y}_\mathrm{M} := \mathrm{Main}(\mathrm{DA}(\mathbf{x}_1) \oplus \mathbf{t}_\mathrm{S}; \mathbf{w}_\mathrm{M})$
 6:     $\mathcal{L}_\mathrm{all} := \mathcal{L}_\mathrm{M}(\mathbf{y}_\mathrm{M}, \mathbf{t}_\mathrm{M})$
 7:     $\mathbf{w}_\mathrm{M} := \mathbf{w}_\mathrm{M} - \alpha\left(\frac{\partial \mathcal{L}_\mathrm{all}}{\partial \mathbf{w}_\mathrm{M}}\right) - \lambda \mathbf{w}_\mathrm{M}$
    {Sub-net training}
 8: for $i = 1 \ldots e$ do
 9:   while not converged do
10:     Sample a mini-batch $B$ of size $b$ from $S$.
11:     $\mathbf{y}_\mathrm{S} := \mathrm{Sub}(\mathrm{DA}(\mathbf{x}_2); \mathbf{w}_\mathrm{S})$
12:     $\mathbf{y}_\mathrm{M} := \mathrm{Main}(\mathrm{DA}(\mathbf{x}_1) \oplus \mathbf{y}_\mathrm{S}; \mathbf{w}_\mathrm{M})$
13:     $\mathcal{L}_\mathrm{all} := \mathcal{L}_\mathrm{M}(\mathbf{y}_\mathrm{M}, \mathbf{t}_\mathrm{M})$
14:     $\mathbf{w}_\mathrm{S} := \mathbf{w}_\mathrm{S} - \alpha\left(\frac{\partial \mathcal{L}_\mathrm{all}}{\partial \mathbf{w}_\mathrm{S}}\right) - \lambda \mathbf{w}_\mathrm{S}$
15: return w
```

To apply DTL to the centrifuge mechanism, we pre-train the downstream main-net to work them as a loss function for the sub-net. The DTL algorithm is described in Algorithm 1 and is given in detail below.

First, fix the value of $\mathbf{y}_\mathrm{S}$ to $\mathbf{t}_\mathrm{S}$ for main-net pre-training,

$$\mathbf{y}_\mathrm{M} = \mathrm{Main}(\mathbf{x}_1 \oplus \mathbf{t}_\mathrm{S}; \mathbf{w}_\mathrm{M}). \tag{6}$$

The pre-learning of the main-net is indicated by Equation (5). After the pre-training of the main-net is completed, its parameters are fixed, and the sub-net is trained with the loss function of the main-net, as shown by

$$\mathbf{w}_\mathrm{S} := \mathbf{w}_\mathrm{S} - \alpha\left(\frac{\partial \mathcal{L}_\mathrm{M}(\mathbf{y}_\mathrm{M}, \mathbf{t}_\mathrm{M})}{\partial \mathbf{w}_\mathrm{S}}\right) - \lambda \mathbf{w}_\mathrm{S}. \tag{7}$$

When we put $\mathcal{L}'_\mathrm{S}(\mathbf{y}_\mathrm{S}) \equiv \mathcal{L}_\mathrm{M}(\mathrm{Main}(\mathbf{x} \oplus \mathbf{y}_\mathrm{S}; \mathbf{w}_\mathrm{M}), \mathbf{t}_\mathrm{M})$, Equation 7 can be expressed as

$$\mathbf{w}_\mathrm{S} := \mathbf{w}_\mathrm{S} - \alpha\left(\frac{\partial \mathcal{L}'_\mathrm{S}(\mathbf{y}_\mathrm{S})}{\partial \mathbf{w}_\mathrm{S}}\right) - \lambda \mathbf{w}_\mathrm{S}. \tag{8}$$

Because the weights of the main-net are fixed, the shape of the $\mathcal{L}'_\mathrm{S}$ function is constant during learning. In other words, $\mathcal{L}'_\mathrm{S}$ serves as a loss function from the standpoint of the sub-net.

B. FINE-TUNING

Fine-tuning [5] modifies the weights of an existing model to train a new task. Output layers are usually extended with randomly initialized weights for the new task. A small learning rate is then used to tune all parameters from their original values to minimize the loss of the new task. Part of the network is sometimes frozen to prevent overfitting.

Fine-tuning, like transfer learning, also classifies the learned model into upstream fine-tuning and downstream fine-tuning depending on the location to which it is applied.

1) UPSTREAM FINE-TUNING: PRE-TRAINING SUB-NET

The same as for UTL, the sub-net is pre-trained according to Equation (4). After completing the pre-training, update the model's parameters using the loss function for the main-net:

$$\mathbf{w} := \mathbf{w} - \alpha\left(\frac{\partial \mathcal{L}_{\mathrm{M}}(\mathbf{y}_{\mathrm{M}}, \mathbf{t}_{\mathrm{M}})}{\partial \mathbf{w}}\right) - \lambda \mathbf{w}, \tag{9}$$

where the parameters of the model are $\mathbf{w} = \{\mathbf{w}_{\mathrm{S}}, \mathbf{w}_{\mathrm{M}}\}$.

2) DOWNSTREAM FINE-TUNING: PRE-TRAINING SUB-NET

The same as for DTL, the main-net is pre-trained according to Equation (5). After completing the pre-training, update the parameters of the model with Equation (9).

C. 2LF: LEARNING WITH TWO LOSS FUNCTIONS

We next describe a method for incorporating both the loss function for the main-net and the loss function for the sub-net into the overall model loss function. There are various methods of incorporation, but the simplest is given by

$$\mathcal{L}_{\mathrm{all}} = \mathcal{L}_{\mathrm{M}}(\mathbf{y}_{\mathrm{M}}, \mathbf{t}_{\mathrm{M}}) + \beta \mathcal{L}_{\mathrm{S}}(\mathbf{y}_{\mathrm{S}}, \mathbf{t}_{\mathrm{S}}), \tag{10}$$

where β is a hyperparameter whose value determines whether the classification accuracy of the main-predictor or the sub-predictor is preferred. However, the appropriate value of β depends on other training conditions, such as the distribution and the volume of the training data set.

D. DTL+ (PROPOSED METHOD)

Fine-tuning updates the parameters of the entire network, and therefore tends to have higher classification performance if it can be trained appropriately compared to transfer learning, which updates only a part of the parameters of the network. However, for example, immediately after the main-net has finished pre-training and moved into the fine-tuning phase in DFT, $\mathbf{t}_{\mathrm{S}}$ which is used for a part of input of main-net will be replaced to the output of the sub-net. When initial values of the sub-net are random values, the value of the network loss function might jump compared to that of the terminal phase during pre-training. This jump sometimes causes catastrophic forgetting [32] in the main-net, making the main-net impossible to propagate to the sub-net the capabilities acquired by the main-net during pre-training. Hence, the jump in the loss function might lead to catastrophic forgetting of the main-net in fine-tuning. For deceasing the influence of the jump, we propose DTL+ which pre-trains the sub-net after pre-training the main-net. DTL+ use DTL for pre-training before fine-tuning, and the details of DTL+ are described in Algorithm 2. We can minimize the

Algorithm 2 DTL+ Algorithm

```
Input: Data augmentation function DA, Loss function $\mathcal{L}_{\mathrm{M}}$,
    main-net predictor Main, sub-net predictor Sub, training
    dataset $S := \bigcup_{i=1}^{n}\{(\mathbf{x}_{1i}, \mathbf{x}_{2i}, \mathbf{t}_{\mathrm{S}i}, \mathbf{t}_{\mathrm{M}i})\}$, mini-batch size
    $b$, weight decay coefficient $\lambda$, scheduled learning rate $\alpha$,
    initial weight $\mathbf{w}_0$, epochs $e$.
Output: Trained weight $\mathbf{w} = \{\mathbf{w}_{\mathrm{S}}, \mathbf{w}_{\mathrm{M}}\}$
    {DTL training}
1: $\mathbf{w}$ = DTL.
    {Fine-tuning}
2: for $i = 1 \ldots e$ do
3:     while not converged do
4:         Sample a mini-batch $B$ of size $b$ from $S$.
5:         $\mathbf{y}_{\mathrm{S}} := \mathrm{Sub}(\mathrm{DA}(\mathbf{x}_2); \mathbf{w}_{\mathrm{S}})$
6:         $\mathbf{y}_{\mathrm{M}} := \mathrm{Main}(\mathrm{DA}(\mathbf{x}_1) \oplus \mathbf{y}_{\mathrm{S}}; \mathbf{w}_{\mathrm{M}})$
7:         $\mathcal{L}_{\mathrm{all}} := \mathcal{L}_{\mathrm{M}}(\mathbf{y}_{\mathrm{M}}, \mathbf{t}_{\mathrm{M}})$
8:         $\mathbf{w} := \mathbf{w} - \alpha\left(\frac{\partial \mathcal{L}_{\mathrm{all}}}{\partial \mathbf{w}}\right) - \lambda \mathbf{w}$
9: return w
```

jumps by moving the output of sub-net closer to $\mathbf{t}_{\mathrm{S}}$ before fine-tuning. The properties of each learning method described above will be discussed in detail in later experiments, and DTL+ is hybrid method of DFT and DTL.

IV. EXPERIMENTAL RESULTS

Given that CPR is a motivation for our research, here we applied the centrifuge mechanism to CPR and observed its effectiveness in experiments.

There are three main objectives of the experiment:

- To confirm the performance and characteristics of each of the normal learning method without any innovations and the six learning methods considered for the centrifuge mechanism described in Section III (Ex. 1 to 3.)
- To confirm the performance and characteristics of the centrifuge mechanism when the number of sub-nets is increased from one to two (Ex. 4.)
- To compare the performance of existing methods (containing BinProv, a state-of-the-art method in CPR) with that of our method (Ex. 5 and 6.)

The source code and dataset used in the experiment are published on GitHub (see DATA Availability Section.)

A. EXPERIMENTAL MODEL

In this section, we describe the model used in our experiments. An overview of the experimental model is shown in Figure 2. We selected self-centrifuge as the centrifuge mechanism and o-glassesX [27] as the model used for the main-net and sub-net. The reasons for selecting each are described below.

1) O-GLASSESX

o-glassesX, described in Section II, has so far shown the best performance in the area of CPR. However, o-glassesX

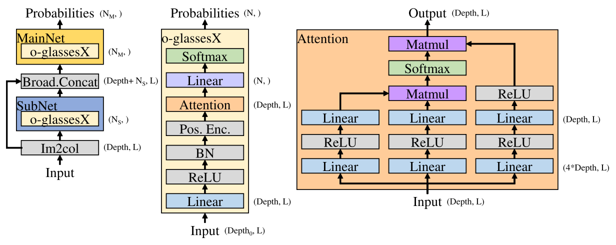


FIGURE 3. Outline of our experimental model.

assumes a single CPU architecture as the classification target since it performs preprocessing according to that architecture. Specifically, preprocessing converts the input bit-stream into a fixed-length machine language instruction sequence, and a tool called a disassembler for each CPU architecture is essential for this conversion. In our experiments, we replaced this CPU architecture-specific preprocessing with the centrifuge mechanism and used raw bit-streams as input.

2) SELF-CENTRIFUGE

Some recent malware attacks do not save an executable file but instead deploy the executable code in a temporary memory area, making it difficult to trace the attacker [13], [14]. In such cases, only a small portion of the executable file is available, and it is necessary to investigate the traces of the attacker from the file fragments. We decided to select the self-centrifuge, which inputs the same program code to both the target and source to allow compiler identification even from file fragments.

On the other hand, if the entire executable file is available, the source-target centrifuge, where the file header area is input to the target ($\mathbf{x}_2$) and the program code to the source ($\mathbf{x}_2$), is considered appropriate. The header area of the executable file contains various metadata, including information about the compiler and CPU architecture. The compiler information is not necessary for executing the executable. An attacker can easily tamper with the compiler information, while the CPU architecture information is essential for executing the executable and is difficult to tamper with.

Figure 3 shows an example of the behavior of the centrifuge mechanism on the learning program code of various CPU architectures. The centrifuge mechanism consists of a sub-net that learns global features and a broadcast concatenator. The broadcast concatenator concatenates an input of the sub-net and the $\mathbf{y}_S$(the sub-net's output.) This concatenation changes operations that use local features of the main net to those that consider the CPU architecture.

B. DATASET

To our knowledge, no dataset for CPR supports multiple CPU architectures. In the area of function similarity comparisons, several large datasets covering multiple CPU architectures have been published [11], [19]. These datasets consist of executable files created by various compilers. However, the executable files contain many functions that do not originate from the compiler and source code files used to make them, such as static libraries. Therefore, it is necessary to remove unnecessary functions from the training dataset for CPR. Extracting only the author-derived functions from the executable files is a research topic, and at this time we are not able to perform this task with complete accuracy.

Therefore, we created the dataset ourselves based on the dataset used in the o-glassesX [27] experiments for enabling performance comparisons with existing methods that support only x86/x86-64. The o-glassesX dataset is a compiler identification dataset and consists mainly of binaries for x86/x86-64 architectures. First, we prepared the same C/C++ source code files from the o-glassesX dataset. We then compiled these source code files with various compiler settings for multiple CPU architectures. Next, we extracted various bits of CPU architecture native code from the generated object files using `elf_coff2bin.py`, which was published with o-glassesX. The size of our dataset was about 1.6 GB (see Appendix for details.)

C. EXPERIMENTAL SETTINGS

We implemented our experimental model in Python and used the PyTorch 1.11.0+cu102 framework on one machine with a single NVIDIA V100 GPU. Each experiment typically took several hours to complete.

We confirmed the performance of our experimental model using the dataset described in the previous section. To have the same number of samples for each label, we set (S) as an upper limit on the number of random samples from each label used in the evaluation experiment.

In order to train a deep model, we use Data Augmentation, a data-space solution to the problem of limited data. The training data generator selects (a) the raw training data or (b) generating the augmented data with a 50-50 probability. When the generator selects (b), then it chooses 15% of the byte positions at random for Data Augmentation. If the i-th byte is chosen, we replace the i-th byte with (1) a random byte 80% of the time (2) the unchanged i-th byte 20% of the time. Then, the augmented data will be used to training the model.

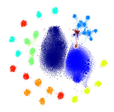
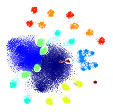
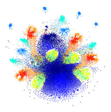
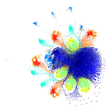
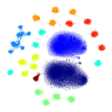
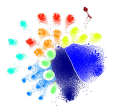
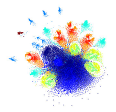
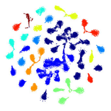
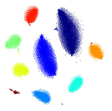
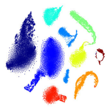
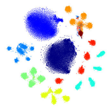
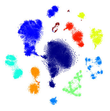
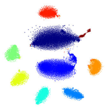
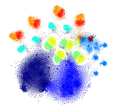
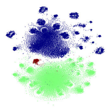
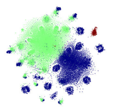
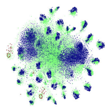
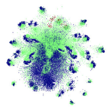
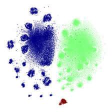
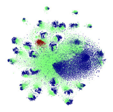
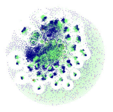
Our experiments had hyperparameters to be tuned, so we first considered the values in the follows when searching for appropriate values of them, and underlines indicate adopted values.

- Label smoothing: none or $\{0.05, \underline{0.1}, 0.2, 0.3\}$
- Initial LR: $\{0.005, 0.01, \ldots, \underline{0.025}, \ldots, 0.05\}$
- Momentum: $\underline{0.9}$
- Weight decay: $\{1\text{e-}5, 5\text{e-}5, \underline{1\text{e-}4}, 5\text{e-}4\}$

Cosine learning rate decay [17] was adopted with an initial learning rate of 0.025. Label smoothing [24] was also adopted with a factor of 0.1.

The other experimental conditions were identical to those employed in o-glassesX, allowing for comparison with existing methods; accuracies were obtained using 4-fold

TABLE 1. Comparison of learning methods for the centrifuge mechanism.

Learning method	UTL	DTL	UFT	DFT	2LF	DTL+	baseline
Pretrain	Sub w/ $\mathcal{L}_S$	Main w/ $\mathcal{L}_M$	Sub w/ $\mathcal{L}_S$	Main w/ $\mathcal{L}_M$	none	DTL	none
Train	Main w/ $\mathcal{L}_M$	Sub w/ $\mathcal{L}_M$	Both w/ $\mathcal{L}_M$	Both w/ $\mathcal{L}_M$	Both w/ $\mathcal{L}_M$&$\mathcal{L}_S$	Both w/ $\mathcal{L}_M$	Both w/ $\mathcal{L}_M$
Sub-nets contribution	unclear	**clear**	unclear	unclear	unclear	**clear**	unclear
Ex.1: Main (51:compiler with opt. lv.), Sub (9:CPU architecture)							
Main Acc.	$97.30_{\pm 0.06}$	$97.17_{\pm 0.11}$	$97.36_{\pm 0.07}$	**97.36**$_{\pm 0.04}$	$97.32_{\pm 0.06}$	$97.35_{\pm 0.04}$	$97.31_{\pm 0.06}$
Sub Acc.	$99.72_{\pm 0.01}$	$99.68_{\pm 0.08}$	$74.74_{\pm 31.11}$	$33.50_{\pm 21.88}$	**99.74**$_{\pm 0.01}$	$96.41_{\pm 4.76}$	$8.73_{\pm 2.98}$
$\mathbf{x}'$ (CPU Arch.)							
Ex.2: Main (9:CPU architecture), Sub (51:compiler with opt. lv.)							
Main Acc.	$99.80_{\pm 0.00}$	$99.76_{\pm 0.03}$	**99.84**$_{\pm 0.01}$	$99.71_{\pm 0.03}$	$99.83_{\pm 0.01}$	$99.78_{\pm 0.01}$	$99.73_{\pm 0.01}$
Sub Acc.	**97.34**$_{\pm 0.02}$	$16.85_{\pm 2.54}$	$19.88_{\pm 1.60}$	$3.39_{\pm 0.85}$	**97.34**$_{\pm 0.03}$	$18.92_{\pm 0.86}$	$1.27_{\pm 1.21}$
Arch. Acc.	$99.78_{\pm 0.01}$	$99.73_{\pm 0.04}$	$99.32_{\pm 0.07}$	$25.97_{\pm 9.87}$	**99.80**$_{\pm 0.01}$	$99.42_{\pm 0.07}$	$10.71_{\pm 11.94}$
$\mathbf{x}'$ (CPU Arch.)							
Ex.3: Main (26:compiler without opt. lv.), Sub (3:opt. lv.)							
Main Acc.	$97.10_{\pm 0.04}$	$96.96_{\pm 0.13}$	$97.01_{\pm 0.05}$	$97.09_{\pm 0.12}$	**97.12**$_{\pm 0.03}$	$97.09_{\pm 0.12}$	$96.92_{\pm 0.14}$
Sub Acc.	$99.70_{\pm 0.01}$	$97.25_{\pm 0.81}$	$51.20_{\pm 3.37}$	$28.36_{\pm 19.45}$	**99.71**$_{\pm 0.03}$	$49.35_{\pm 0.43}$	$39.18_{\pm 13.91}$
$\mathbf{x}'$ (Opt. lv.)							

The best results under the same conditions are indicated in **bold**, and the worst results are denoted by underlined text. $\mathbf{x}'$ is the output of the main-net first layer visualized by tSNE, and color-coding rules are described in parentheses following $\mathbf{x}'$. "Arch. Acc." is the accuracy of the sub-net when the grand truth and the predictions are grouped by CPU architecture. "Sub-nets contribution" indicates whether or not the output of the main-net is learning under the control of the output of the sub-net. The concept of "Sub-nets contribution" was described in Section III.

cross-validation, and the other parameter configurations were as follows.

- input length (L) = 235 bytes
- S (Num. of random samples from each label) = 20000
- mini-batch size = 64
- epochs = 200

These parameters were set in order to make comparisons under conditions equivalent to those of o-glassesX, an existing study. Since x-86/x-86-64 are variable-length instructions, and the input size of 64 instructions, which is the input size of o-glassesX, is just under 236 bytes on average, the input size of our model was set 235 bytes. In other words, similar to o-glassesX, our method identifies the compiler from program fragments.

D. EX. 1 TO EX. 3: COMPARISON BETWEEN THE SIX LEARNING METHODS

In this section, we check the classification performance of each learning method using three different main-labels and sub-labels for our dataset. Each labeling rule is as shown in "Label" in Table 1, specifically the following three labeling rules.

- Ex. 1: We set main labels for 51 classes of compiler identification, including optimization level identification, and sub-labels for nine classes of CPU architecture identification. We then check the classification performance of our experimental model when the sub-prediction is easier than the main prediction.
- Ex. 2: We swapped the main-labels and sub-labels from Ex. 1. We then check the classification performance of our experimental model with nine main-labels for the CPU architecture and 51 sub-labels for compiler identification with optimization level identification included.
- Ex. 3: We check the classification performance of our experimental model when the main-label and sub-label are independent. We chose 26 labels for compiler identification that did not include optimization level estimation as main-labels.

Table 1 shows a comparison of the learning methods for the centrifuge mechanism and baseline, which simply trains both the main-net and sub-net with $\mathcal{L}_M$.

The $\mathbf{x}'$ in Table 1 shows the main-net first linear layer output visualized by tSNE [39], color-coded by CPU architectures or optimization levels. $\mathbf{x}'$ is indicated by Equation 3, and we expect $\mathbf{x}'$ to be the input $\mathbf{x}$ transitioned by each sub-label, as shown in Figure 2.

Simply comparing the accuracy of the main-label classification, we can see no significant performance difference among any of the learning methods. On the other hand, when we focus on the accuracy of sub-label classification, we find that fine-tuning does not perform well in sub-label classification due to catastrophic forgetting [22].

Focusing on DTL in Ex. 1 and Ex. 3, we can see that the sub-prediction achieves a remarkable degree of accuracy, even though only the loss function for the main-label is used.

In Ex. 2, the sub-predictions are not well classified except for UTL and 2LF. One possible cause is that one main-label has a relationship that encompasses multiple sub-labels. If the main label classification is known to belong to one of those multiple sub-labels, then it can be classified without identifying the difference between them. Therefore, it is likely that error propagation from the main-net to the sub-net did not work to improve the accuracy of sub-label classification. From the main classification standpoint, the accuracy of the sub-label classification does not need to be perfect, it only needs to be correct at the CPU architecture level for the main-label. DTL's $\mathbf{x}'$ of Ex. 2 in the table clearly shows that $\mathbf{x}'$ is limited to the essence for the main prediction, that is, CPU architecture identification. Therefore, the sub-net accuracy of DTL may measure the importance of sub-label classification for main-label classification.

In summary, focusing on DTL in particular, we found the following characteristics compared with other learning methods.

- DTL can incorporate highly accurate sub-predictions into the model without having to adjust hyperparameters or design loss functions for sub-predictions, such as 2LF's β.
- The accuracy of DTL sub-predictions depends on the essence of the main forecast, and it might be used to measure their contribution to the main prediction.

Finally, focusing on DTL+, we see that its main-net classification performance is exactly halfway between that of DTL and DFT. In addition, the sub-net classification performance shows that DTL+ suppresses catastrophic forgetting considerably while other fine-tuning methods cause catastrophic forgetting. Therefore, DTL+ is considered to inherit the property of DTL that it learns the main-net according to the domain (CPU architecture) in which the subnet is determined. Thus, we experimentally confirmed that DTL+ can realize a hybrid method of DTL and DFT.

E. EX. 4: CLASSIFICATION PERFORMANCE IN THE CASE OF A MODEL CONTAINING TWO SUB-NETS

The results of Ex. 1 to Ex. 3 were those for which the model contained a single sub-net. This section will examine the case where the number of sub-nets is increased to two. As with the other experiments, o-glassesX was used for the main-net and the two sub-nets. The main-labels are for compiler identification with optimization level identification, and there are 51 of them. The first sub-labels are three labels for optimization level identification. The second sub-labels are nine labels for CPU architecture identification. The output of the two sub-nets and the input $\mathbf{x}$ are combined and put into the main-net.

Based on the results of Ex. 1 to Ex. 3, we selected DTL and DTL+ as the learning method since there is no need to design a loss function for sub-predictions.

TABLE 2. The results of Ex. 4.

	Main	Sub$_1$	Sub$_2$
DTL*	$97.07_{\pm 0.08}$	$99.70_{\pm 0.06}$	$99.71_{\pm 0.01}$
DTL+*	$97.68_{\pm 0.04}$	$97.47_{\pm 0.14}$	$96.26_{\pm 4.54}$
DTL	$96.67_{\pm 0.71}$	$87.03_{\pm 21.95}$	$92.27_{\pm 12.88}$
DTL+	$97.68_{\pm 0.08}$	$85.72_{\pm 20.35}$	$95.59_{\pm 4.10}$

*: The values exclude the results of one of the four-part cross-validation experiments, which failed to learn and significantly degraded performance.

Table 2 shows the results of Ex. 4. The experimental results show that even when the number of sub-nets is increased to two, sub-predictions can be made with high accuracy in DTL and DTL+. Thus, it may be easier to develop a model with more complex coordination of sub-nets.

F. PERFORMANCE COMPARISON WITH EXISTING METHODS

1) EX. 5: SINGLE CPU ARCHITECTURE (X86/X86-64)

In this section, we compare our proposed and existing methods' classification accuracy. We have to compare our proposed method with existing CPR methods that support multiple CPU architectures. In most studies, the classification targets are programs with an x86/x86-64 architecture. Therefore, we conducted a comparison experiment using only the x86/x86-64 architecture programs among our dataset (only those marked with a checkmark in Table 5.) We set three architectures (x86/x86-64/Others) as sub-labels in our method and selected DFT as the learning method. Table 3 shows a comparison between our method and other existing methods.

Table 3 shows that our model is lightweight and our method achieved the best classification performance in both existing methods with and without DE. In Table 3, "CNN" has the same model as o-glasses' but does not perform pre-processing for instruction segmentation. For existing methods that cannot be tested on our dataset due to different feature vectors, the classification accuracies are taken from their papers.

The only difference between CNN and o-glasses is whether or not DE is performed. However, there are significant differences in performance between the two methods. This difference in performance indicates that DE plays a very important role in CPR. On the other hand, our method without DE outperforms o-glassesX with DE. Therefore, we believe that the centrifuge mechanism can replace DE. By not relying on DE, the model's range of applicability may be expanded, because it can be trained on a mixed data set such as ours.

2) EX. 6: MULTI-CPU ARCHITECTURE

We compared our method with existing CPR methods for multi-CPU architectures. To the best of our knowledge, the only relevant existing work is BinProv, so we compared our method with BinProv. There are two types of BinProv: BinProv w/o, which performs compiler identification from

TABLE 3. Performance comparison with related work (on x86/x86-64).

Task	Method	Acc.		ML model	No DE	Input	#Labels	Model size
Nine compilers, optimization(High/Low) and others	Ours (DTL+)	**99.01**		Attention	✓	235 bytes	19	1M
	o-glassesX [27]	98.84	[27]	Attention		64 Instr.[a]	19	0.5M
	o-glasses [26]	94.21	[27]	CNN		64 Instr.[a]	19	6.5M[b]
	CNN	36.33		CNN	✓	235 bytes	19	6.5M[b]
Two compilers and optimization(High/Low)	Ours (DTL+)	**99.17**		Attention	✓	235 bytes	4	1M
	o-glassesX [27]	96.15	[6]	Attention		16 Instr.	4	0.5M
	o-glassesX* [6]	75.16	[6]	Attention	✓	256 bytes	4	0.5M
	BinProv w/o [6]	94.77	[6]	RoBERTa	✓	512 bytes	4	355M
	BinRNN [6]	64.64	[6]	RNN	✓	512 bytes	4	negl[c]
	ORIGIN [33]	93.92	[6]	(Unknown)		(Unknown)	4	negl[c]
Other tasks	Rosenblum's [35]	92.4	[35]	CRF		Instr. seq.	3	negl[c]
	ORIGIN (SVM) [33]	60.4	[33]	SVM		1 func	18	negl[c]
	ORIGIN (CRF) [33]	91.8	[33]	CRF		Func seq.	18	negl[c]
	BinComp [31]	80.1	[31]	(k-means)		1 file	6	negl[c]

The best result is indicated in **bold**. [a]: The average size of 64 instructions is 236 bytes. When there is a citation of a paper to the right of "Acc.", it indicates that the experimental results are taken from these papers. [b]: The model size increases as the number of input instructions increases, since the input data is amplified in the CNN layer and then input to the fully connected layer. [c]: The parameters of the model itself are shown as "negl" because they are negligibly small compared to other methods.

TABLE 4. Performance comparison with BinProv (on multi-CPU Architecture).

	Compiler(G/C)		Opt level (H/L)		Overall (Basic task)	
Setting	BinProv w/o	Ours	BinProv w/o	Ours	BinProv w/o	Ours
x86_64	95.47	99.15	98.90	99.93	94.77	99.17
x86_32	96.22	99.41	-	99.92	-	99.39
ARM_64	98.80	99.77	-	99.91	-	99.69
MIPS_64	98.55	99.82	-	99.97	-	99.80
All	-	99.54	-	99.93	-	99.43

The experimental results of BinProv are quoted from the paper [6]. Experimental results not mentioned in the paper are indicated by "–".

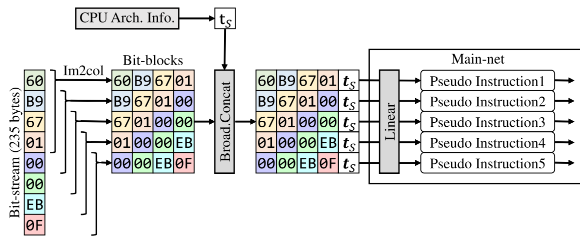
a single file fragment, and BinProv, which aggregates the multiple identification results and performs estimation on a file or function basis. Since the aggregation process is a simple majority vote, the classification accuracy from a single file fragment directly affects the classification accuracy per file or function. Therefore, we chose BinProv w/o for comparison with our method. Table 4 shows a comparison between our method and BinProv w/o.

BinProv has a project on GitHub at the time of writing our paper, but the source code has not been uploaded, so we quoted the experimental results described in their paper. Table 4 shows that our method has better classification performance than BinProv.

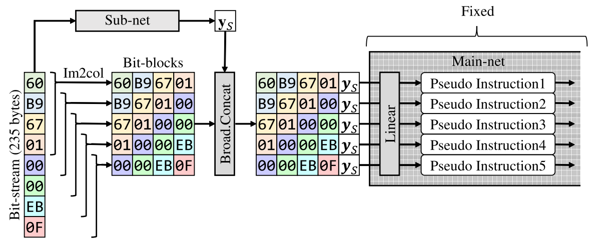
V. DISCUSSION

A. THE OPTIMAL LEARNING METHOD

The optimal learning method depends on the specific goal. Ex. 1's classification accuracy is displayed in Figure 4. If the primary objective is classification accuracy for the main category, Ex. 1 showed that DFT had the highest performance, although there may not be a significant difference in the classification performance of each method. However, if the focus is on subcategory classification performance, UTL, DTL, 2LF, and DTL+ demonstrated excellent performance. Of the four learning methods mentioned, only DTL+ can be trained using solely the loss function for the main category.



(a). Main-net training with domain information like the CPU architecture. In the main-net, pseudo-instructions are generated that take the CPU architecture into consideration.



(b). Sub-net training with fixed main-net parameters. If $\mathbf{t_S}$ is an important element for the main-net, the sub-net parameters are updated so that the value of $\mathbf{y_S}$, which is the output of the sub-net, approaches the value of $\mathbf{t_S}$.

FIGURE 4. An example of the behavior of the centrifuge mechanism on the learning program code of various CPU architectures.

This makes DTL+ superior as it does not necessitate the design of loss functions for subcategories and can be widely applied.

B. BENEFITS OF OUR METHOD

1) HIGH RECOGNITION RATE FOR STRIPPED MACHINE CODE

Our approach can be applied even when symbol information has been stripped or is otherwise unavailable, because our method classifies code sequences instead of entire binaries. In other words, our method relies solely on characteristics of the binary code rather than metadata or other program header details. Therefore, it can be used even when code produced

TABLE 5. Overview of our dataset.

							Label												
							Ex.1		Ex.2		Ex.3		Ex.4			Ex.5		Ex.6	
Arch.	Compiler	Opt.	Binaries	Size (MB)	Entropie	(1)	M	S	M	S	M	S	M	S1	S2	M	S	M	S
x86	VC2003	-Od	1,350	16.24	5.54	✓	0	0	0	0	0	0	0	0	0	0	0		
		-Ox	1,306	18.96	6.03	✓	1	0	0	1	0	1	1	1	0	1	0		
	VC2017	-Od	1,170	18.96	5.35	✓	2	0	0	2	1	0	2	0	0	2	0		
		-Ox	1,147	14.61	5.88	✓	3	0	0	3	1	1	3	1	0	3	0		
	gcc 6.3.0	-O0	2,111	17.20	5.85	✓	4	0	0	4	2	0	4	0	0	4	0	0	0
		-O3	1,844	18.04	6.14	✓	5	0	0	5	2	1	5	1	0	5	0	1	0
	gcc 7.5.0	-O0	3,006	30.86	5.63		6	0	0	6	3	0	6	0	0			0	0
		-O3	2,756	34.18	6.10		7	0	0	7	3	1	7	1	0			1	0
	clang 5.0.2	-O0	1,205	9.24	5.77	✓	8	0	0	8	4	0	8	0	0	6	0	2	0
		-O3	1,196	8.43	6.23	✓	9	0	0	9	4	1	9	1	0	7	0	3	0
	clang 10.0.0	-O0	3,258	28.55	5.57		10	0	0	10	5	0	10	0	0			2	0
		-O3	3,278	27.45	6.12		11	0	0	11	5	1	11	1	0			3	0
	ICC	-Od	1,761	76.39	5.45	✓	12	0	0	12	6	0	12	0	0	8	0		
		-Ox	1,724	66.56	5.69	✓	13	0	0	13	6	1	13	1	0	9	0		
x86-64	VC2017	-Od	1,456	39.91	4.79	✓	14	1	1	14	7	0	14	0	1	10	1		
		-Ox	1,242	32.37	5.83	✓	15	1	1	15	7	1	15	1	1	11	1		
	gcc 6.3.0	-O0	1,582	20.70	5.46	✓	16	1	1	16	8	0	16	0	1	12	1	4	1
		-O3	1,580	22.68	6.15	✓	17	1	1	17	8	1	17	1	1	13	1	5	1
	gcc 7.5.0	-O0	3,058	31.00	5.20		18	1	1	18	9	0	18	0	1			4	1
		-O3	2,804	33.94	6.04		19	1	1	19	9	1	19	1	1			5	1
	clang 5.0.2	-O0	1,892	26.23	5.25	✓	20	1	1	20	10	0	20	0	1	14	1	6	1
		-O3	1,883	20.80	6.13	✓	21	1	1	21	10	1	21	1	1	15	1	7	1
	clang 10.0.0	-O0	3,063	31.42	5.24		22	1	1	22	11	0	22	0	1			6	1
		-O3	3,062	29.95	5.98		23	1	1	23	11	1	23	1	1			7	1
	ICC	-Od	1,796	89.89	4.90	✓	24	1	1	24	12	0	24	0	1	16	1		
		-Ox	1,728	72.03	6.00	✓	25	1	1	25	12	1	25	1	1	17	1		
Armv7	gcc 7.5.0	-O0	2,493	21.27	6.18		26	2	2	26	13	0	26	0	2				
		-O3	2,375	21.04	7.12		27	2	2	27	13	1	27	1	2				
	clang 10.0.0	-O0	3,249	28.11	5.07		28	2	2	28	14	0	28	0	2				
		-O3	3,249	27.82	5.77		29	2	2	29	14	1	29	1	2				
Arm64	gcc 7.5.0	-O0	2,501	25.68	5.35		30	3	3	30	15	0	30	0	3			8	2
		-O3	2,382	27.48	6.39		31	3	3	31	15	1	31	1	3			9	2
	clang 10.0.0	-O0	3,257	27.23	5.90		32	3	3	32	16	0	32	0	3			10	2
		-O3	3,257	26.40	6.41		33	3	3	33	16	1	33	1	3			11	2
MIPS	gcc 7.5.0	-O0	2,494	35.95	4.26		34	4	4	34	17	0	34	0	4				
		-O3	2,376	31.20	5.66		35	4	4	35	17	1	35	1	4				
	clang 10.0.0	-O0	3,243	33.81	4.50		36	4	4	36	18	0	36	0	4				
		-O3	3,243	31.08	5.09		37	4	4	37	18	1	37	1	4				
MIPS64	gcc 7.5.0	-O0	2,451	36.11	4.56		38	5	5	38	19	0	38	0	5			12	3
		-O3	2,333	30.60	5.50		39	5	5	39	19	1	39	1	5			13	3
	clang 10.0.0	-O0	3,203	39.44	4.58		40	5	5	40	20	0	40	0	5			14	3
		-O3	3,203	32.61	5.32		41	5	5	41	20	1	41	1	5			15	3
PowerPC	gcc 7.5.0	-O0	2,491	28.95	5.52		42	6	6	42	21	0	42	0	6				
		-O3	2,373	30.00	6.12		43	6	6	43	21	1	43	1	6				
	clang 10.0.0	-O0	3,246	29.24	4.89		44	6	6	44	22	0	44	0	6				
		-O3	3,246	30.03	5.40		45	6	6	45	22	1	45	1	6				
PowerPC64	gcc 7.5.0	-O0	2,450	32.69	5.44		46	7	7	46	23	0	46	0	7				
		-O3	2,332	34.39	6.05		47	7	7	47	23	1	47	1	7				
	clang 10.0.0	-O0	3,203	30.79	5.29		48	7	7	48	24	0	48	0	7				
		-O3	3,203	32.39	5.84		49	7	7	49	24	1	49	1	7				
Others			101	33.34	7.65	✓	50	8	8	50	25	2	50	2	8	18	2		
Total			120,212	1,594.24	5.65														

"(1) "is the same subset as a subset of the dataset of o-glassesX [27]. "Label" in this table indicates the labels used in the five experiments described below. "M" is the main label, and "S" is the sub-label.

by multiple compilers coexists within a program binary, such as statically linked library code.

2) LIGHTWEIGHT MODEL

Although the model used in the experiment is significantly smaller than state-of-the-art models, it demonstrated excellent classification performance in all tasks, which may be attributed to the size of the training dataset. It is widely acknowledged that increasing the training dataset and computational cost can enhance model performance in line with the model size. However, it is important to note that if the training dataset is not sufficiently large despite increasing the model size, overfitting may occur, which could lead to a decrease in the model's generalization performance. Therefore, if the training dataset is large enough, it is possible that BinProv will outperform the proposed method in terms of classification performance. However, it may be worth considering the cost of collecting training datasets and calculation expenses when evaluating the proposed method's applicability. It is believed that the proposed method may have a broader range of applicability.

C. LIMITATION

Our method assumes that the machine code is not obfuscated. Therefore, the classification performance of our method when the machine code is compressed or obfuscated

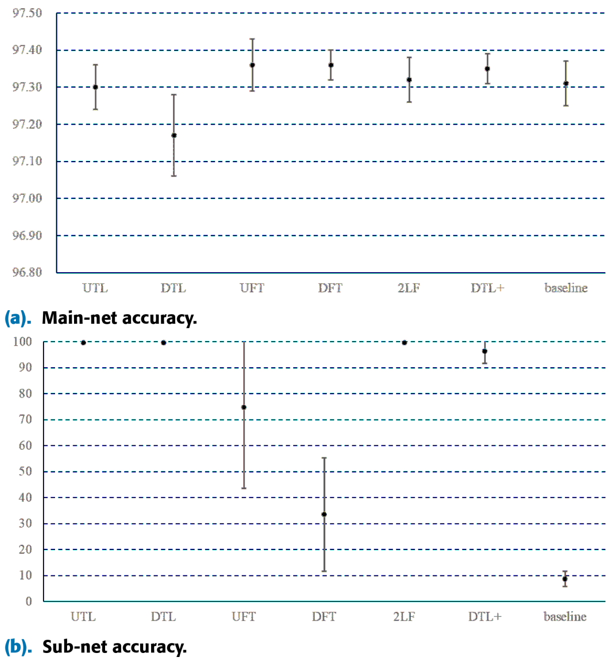


FIGURE 5. Classification accuracy of each learning method in Ex.1 (see Table 1 for details).

is unknown. Experiments have confirmed that BinProv performs well for simple obfuscations such as instruction substitution [6]. However, they claim that multiple obfuscations are very difficult to classify, and our method is expected to be very difficult as well.

VI. CONCLUSION

We proposed a centrifuge mechanism, in which the upstream sub-net transitions the input to a space corresponding to sub-labels without requiring manual DE. This enables the downstream main-net to concentrate more on challenging main-label classifications. DTL, one of the learning methods for the centrifuge mechanism, pre-trains the main-net using the ground truth of the sub-labels instead of the output of the sub-net. DTL was found to have the ability to assign the sub-net the task of sub-label prediction without utilizing a loss function for sub-label classification. Additionally, it was observed that sub-predictions are typically highly accurate when the sub-label classification is essential to the main prediction. The centrifuge mechanism was applied to CPR and its performance was verified in experiments. The experiments have shown that the experimental model without DE was able to classify bit-streams with an accuracy of 99.01. This accuracy is higher than the existing CPR methods with DE for x86/x86-64 CPU architecture. The model was tested for compiler identification for four CPU architectures and two compilers, and despite its smaller size, our method achieved the highest classification performance of 99.54, outperforming Binprov which is the state-of-the-art model. Furthermore, it was found to be able to classify 51 classes with an accuracy of 97.36. It has been observed that our compiler identification model is lightweight and demonstrates superior classification performance for compiler identification when compared to existing methods.

Executable files often contain non-author code, such as static link libraries, which can decrease CPR accuracy. To improve CPR accuracy, future work will investigate technology capable of extracting only author-originated code from executable files.

APPENDIX

Table 5 shows an outline of the dataset. The dataset consists of the native code of each architecture except "Others." "Others" is non-native code data created from various document files (.rtf, .doc, .docx, and .pdf files). "Entropy" in the table is Shannon's entropy [37]. The entropy $H(X)$ is defined as

$$H(X) = -\sum_{i=0}^{255} p(x_i) log_2 p(x_i). \tag{11}$$

Since we calculated Shannon's entropy in bytes, H(X) takes values from 0 to 8, and the larger the value, the more disordered the state. "Label" in Table 5 indicates the labels used in the five experiments described below. "M" is the main label, and "S" is the sub-label.

DATA AVAILABILITY

The research materials are available at the following URL: https://github.com/yotsubo/o-glasses2023.

REFERENCES

[1] T. Benoit, J.-Y. Marion, and S. Bardin, "Binary level toolchain provenance identification with graph neural networks," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2021, pp. 131–141.

[2] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2014, pp. 1724–1734.

[3] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," in *Proc. NIPS Workshop Deep Learn.*, Dec. 2014.

[4] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, Sep. 1995.

[5] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, New York, NY, USA, Jun. 2014, pp. 580–587.

[6] X. He, S. Wang, Y. Xing, P. Feng, H. Wang, Q. Li, S. Chen, and K. Sun, "BinProv: Binary code provenance identification without disassembly," in *Proc. 25th Int. Symp. Res. Attacks, Intrusions Defenses*, Oct. 2022, pp. 350–363.

[7] H.-S. Heo, B.-M. So, I.-H. Yang, S.-H. Yoon, and H.-J. Yu, "Automated recovery of damaged audio files using deep neural networks," *Digit. Invest.*, vol. 30, pp. 117–126, Sep. 2019.

[8] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Institut Informatik Technische Universität München, Tech. Rep. FKI-207-85, 1995.

[9] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[10] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020, *arXiv:2001.08361*.

[11] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 1661–1682, Apr. 2023.

[12] A. Krogh and J. Hertz, "A simple weight decay can improve generalization," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 4, 1991, pp. 950–957.

[13] S. Kumar, U. Dohare, K. Kumar, D. P. Dora, K. N. Qureshi, and R. Kharel, "Cybersecurity measures for geocasting in vehicular cyber physical system environments," *IEEE Internet Things J.*, vol. 6, no. 4, pp. 5916–5926, Aug. 2019.

[14] Sudhakar and S. Kumar, "An emerging threat fileless malware: A survey and research challenges," *Cybersecurity*, vol. 3, no. 1, pp. 1–12, Dec. 2020.

[15] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Comput.*, vol. 1, no. 4, pp. 541–551, Dec. 1989.

[16] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A robustly optimized BERT pretraining approach," 2019, *arXiv:1907.11692*.

[17] I. Loshchilov and F. Hutter, "SGDR: Stochastic gradient descent with warm restarts," 2016, *arXiv:1608.03983*.

[18] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," 2015, *arXiv:1508.04025*.

[19] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *Proc. 31st USENIX Secur. Symp.* Boston, MA, USA: USENIX Association, Aug. 2022, pp. 2099–2116.

[20] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *Proc. Workshop Binary Anal. Res.*, 2019.

[21] A. McCallum, "Efficiently inducing features of conditional random fields," in *Proc. 19th Conf. Uncertainty Artif. Intell.* Vurlington, MA, USA: Morgan Kaufmann, 2002, pp. 403–410.

[22] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," *Psychol. Learn. Motivat.*, vol. 24, pp. 109–165, Jan. 1989.

[23] T. Mikolov, K. Chen, G. S. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. Int. Conf. Learn. Represent.*, 2013.

[24] R. Müller, S. Kornblith, and G. E. Hinton, "When does label smoothing help?" in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 4696–4705.

[25] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: Visualization and automatic classification," in *Proc. 8th Int. Symp. Visualizat. Cyber Secur.* New York, NY, USA: Association for Computing Machinery, Jul. 2011, pp. 1–7.

[26] Y. Otsubo, A. Otsuka, M. Mimura, and T. Sakaki, "O-Glasses: Visualizing x86 code from binary using a 1D-CNN," *IEEE Access*, vol. 8, pp. 31753–31763, 2020.

[27] Y. Otsubo, A. Otsuka, M. Mimura, T. Sakaki, and H. Ukegawa, "O-GlassesX: Compiler provenance recovery with attention mechanism from a short code fragment," in *Proc. Workshop Binary Anal. Res.* Reston, VA, USA: Internet Society, 2020, pp. 1–12.

[28] B. M. Padmanabhuni and H. B. K. Tan, "Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf.*, vol. 2, New York, NY, USA, Jul. 2015, pp. 450–459.

[29] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 10, pp. 1345–1359, Jan. 2009.

[30] D. Pizzolotto and K. Inoue, "Identifying compiler and optimization level in binary code from multiple architectures," *IEEE Access*, vol. 9, pp. 163461–163475, 2021.

[31] A. Rahimian, P. Shirani, S. Alrbaee, L. Wang, and M. Debbabi, "BinComp: A stratified approach to compiler provenance attribution," *Digit. Invest.*, vol. 14, pp. 146–155, Aug. 2015.

[32] V. V. Ramasesh, E. Dyer, and M. Raghu, "Anatomy of catastrophic forgetting: Hidden representations and task semantics," 2020, *arXiv:2007.07400*.

[33] N. Rosenblum, B. P. Miller, and X. Zhu, "Recovering the toolchain provenance of binary code," in *Proc. Int. Symp. Softw. Test. Anal.* New York, NY, USA: Association for Computing Machinery, Jul. 2011, pp. 100–110.

[34] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt, "Machine learning-assisted binary code analysis," in *Proc. NIPS Workshop Mach. Learn. Adversarial Environments Comput. Secur.*, 2007, pp. 1–3.

[35] N. E. Rosenblum, B. P. Miller, and X. Zhu, "Extracting compiler provenance from program binaries," in *Proc. 9th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng.* New York, NY, USA: Association for Computing Machinery, May 2010, pp. 21–28.

[36] N. E. Rosenblum, X. Zhu, and B. Miller, "Who wrote this code? Identifying the authors of program binaries," in *ESORICS 2011*. Berlin, Germany: Springer, 2011, pp. 172–189.

[37] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, no. 3, pp. 379–423, Jul. 1948.

[38] Z. Tian, Y. Huang, B. Xie, Y. Chen, L. Chen, and D. Wu, "Fine-grained compiler identification with sequence-oriented neural modeling," *IEEE Access*, vol. 9, pp. 49160–49175, 2021.

[39] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *J. Mach. Learn. Res.*, vol. 9, no. 86, pp. 2579–2605, 2008.

[40] S. Yang, Z. Shi, G. Zhang, M. Li, Y. Ma, and L. Sun, "Understand code style: Efficient CNN-based compiler optimization recognition system," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2019, pp. 1–6.

[41] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, Jan. 2020.

YUHEI OTSUBO was born in Fukuoka, Japan, in 1981. He received the B.S. degree from The University of Tokyo, Japan, in 2005, the M.S. degree from the National Graduate Institute for Policy Studies, Japan, in 2012, and the Ph.D. degree in informatics from the Institute of Information Security, Kanagawa, Japan, in 2016.

Since 2005, he has been a technical official with the National Police Agency, Japan. From 2012 to 2014, he was with the National Information Security Center. His research interest includes information security. He was a Speaker of Black Hat USA, in 2016.

AKIRA OTSUKA (Member, IEEE) received the Ph.D. degree from The University of Tokyo, in 2002. Since 2005, he has been with the National Institute of Advanced Industrial Science and Technology (AIST), where he was the Leader of Research Security Fundamentals, from 2006 to 2010. From 2007 to 2014, he was a Visiting Professor with the Research and Development Initiative, Chuo University. Since 2017, he has been a Professor with the Graduate School of Information Security, Institute of Information Security. From 2020 to 2021, he was a Visiting Researcher with the Financial Research Institute, Bank of Japan. He is a Senior Member of IEICE and IPSJ, and a member of JSAI, IACR, and IFCA.

MAMORU MIMURA received the B.E. and M.E. degrees in engineering from the National Defense Academy, Japan, in 2001 and 2008, respectively, the Ph.D. degree in informatics from the Institute of Information Security, in 2011, and the M.B.A. degree from Hosei University, in 2014. From 2001 to 2017, he was a member of Japan Maritime Self-Defense Force. From 2011 to 2013, he was with the National Information Security Center. Since 2015, he has been with the National Center of Incident Readiness and Strategy for Cybersecurity. He is currently an Associate Professor with the Department of Computer Science, National Defense Academy, Japan.